



RESEARCH ARTICLE / ARAŞTIRMA MAKALESİ

Quantitative Performance Analysis of BLAS Libraries on GPU Architectures

BLAS Kütüphanelerinin GPU Mimarilerindeki Nicel Performans Analizi

Işıl Öz 

İzmir Yüksek Teknoloji Enstitüsü, Bilgisayar Mühendisliği, İzmir, TÜRKİYE
Corresponding Author / Sorumlu Yazar*: isiloz@iyte.edu.tr

Abstract

Basic Linear Algebra Subprograms (BLAS) are a set of linear algebra routines commonly used by machine learning applications and scientific computing. BLAS libraries with optimized implementations of BLAS routines offer high performance by exploiting parallel execution units in target computing systems. With massively large number of cores, graphics processing units (GPUs) exhibit high performance for computationally-heavy workloads. Recent BLAS libraries utilize parallel cores of GPU architectures efficiently by employing inherent data parallelism. In this study, we analyze GPU-targeted functions from two BLAS libraries, cuBLAS and MAGMA, and evaluate their performance on a single-GPU NVIDIA architecture by considering architectural features and limitations. We collect architectural performance metrics and explore resource utilization characteristics. Our work aims to help researchers and programmers to understand the performance behavior and GPU resource utilization of the BLAS routines implemented by the libraries.

Keywords: Basic linear algebra subprograms, Graphics processing units, Performance analysis

Öz

Temel Lineer Cebir Alt Programları (BLAS), makine öğrenmesi ve bilimsel hesaplama tarafından yaygın olarak kullanılan lineer cebir rutinleri içermektedir. BLAS rutinlerinin optimize edilmiş uygulamalarına sahip BLAS kütüphaneleri, bilgisayar sistemlerindeki paralel yürütme birimlerinden yararlanarak yüksek performans sunmaktadır. Çok sayıda çekirdeğe sahip olan grafik işlemci birimleri, hesaplama açısından ağır iş yükleri için yüksek performans sergilemektedir. Modern BLAS kütüphaneleri, veri paralellliğini kullanarak GPU mimarilerini verimli bir şekilde kullanmaktadır. Bu çalışmada, iki BLAS kütüphanesi (cuBLAS ve MAGMA) fonksiyonları analiz edilmiş, mimari özellikleri ve sınırlamaları göz önünde bulundurularak NVIDIA GPU mimarileri üzerindeki performansları değerlendirilmiştir. Performans metrikleri toplanmış ve kaynak kullanım özellikleri tespit edilmiştir. Çalışmamız, araştırmacıların ve programcıların BLAS rutinlerinin performans davranışını ve GPU kaynak kullanımını anlamalarına yardımcı olmayı amaçlamaktadır.

Anahtar Kelimeler: Temel lineer cebir alt programları, Grafik işlemci birimleri, Performans analizi

1. Introduction

With a massive number of processing units, GPU architectures offer performance improvements for applications from various domains, such as high-performance computing workloads [1-3], deep learning training and inference computations [4,36], safety-critical software for autonomous vehicles [5,35], and avionics systems [6]. Specifically, Deep Neural Networks (DNN) require high computational resources to enable parallel execution for their huge data-parallel operations. Essentially, neural networks include heavy linear algebra operations, mostly matrix multiplications, by considering different layers in the network. GEMM (general matrix multiplication) operation constitutes a substantial part of the convolutional layer computations in different CNN implementations [34,37,38]. Basic Linear Algebra Subprograms (BLAS) libraries offer high-performance vector and matrix operations by utilizing parallel execution units on both CPU and GPU architectures [7]. Deep Learning (DL) frameworks like PyTorch, Caffe, and TensorFlow rely on high-performance BLAS libraries exploiting parallel GPU cores to accelerate the training process [8].

Most BLAS libraries conform to the BLAS interface, revealing standard functions that enable library users to develop programs

without worrying about the implementation-specific details. While offering a generic interface to the developers, BLAS libraries are implemented by considering various design and optimization options. While CUDA-enabled libraries [9,10] like cuBLAS exploit the computational resources of NVIDIA GPUs, there are implementations [11,12] like rocBLAS that are written by HIP programming language and optimized for AMD GPUs. On the other hand, MAGMA (Matrix Algebra for GPU and Multicore Architectures) [13] provides high-level interfaces by supporting different CPU (like Intel MKL (Math Kernel Library) or OpenBLAS) or GPU (like cuBLAS or hipBLAS) backends while it offers its own implementations of some routines for better optimization. Modern BLAS libraries [14] support or are optimized for multi-GPU execution to exploit more parallelism. Additionally, for portability, open-source libraries like CLBlast [15] provide optimized OpenCL (Open Computing Language) routines for a wide variety of devices.

In this work, we analyze two BLAS libraries, including cuBLAS [10] and MAGMA [13], and evaluate their performance on a single-GPU NVIDIA architecture by considering architectural constraints. Although there are studies comparing CPU and GPU performance of BLAS libraries [16,17] or comparing the

performance of GPU-based BLAS libraries for specific problems [18], we evaluate complete GPU-optimized Level-3 and one Level-2 routines with high complexity from both libraries by profiling their architectural features and limitations and analyze the computing efficiency of different subroutines from different libraries.

The main contributions of our work are as follows:

- We conduct a detailed experimental analysis for high-complexity BLAS routines from two BLAS libraries on a modern GPU device.
- We identify the most time-consuming computations, i.e., kernel functions, of the corresponding routines to point out the potential performance exploration.
- We collect architectural performance metrics about the main kernel functions of the corresponding routines.
- Based on our profiling results, we evaluate and compare the performance of the library functions according to the computation and memory resources of the GPU device.

Our main goal is to present a quantitative analysis to researchers and programmers who are involved in scientific computing and developing machine learning systems.

The remainder of this paper is organized as follows: Section 2 presents some background on BLAS routines and GPU architectures. We explain our methodology in Section 3. Then the experimental evaluation is presented in Section 4 and the related discussion is given in Section 5. Section 6 presents the existent performance comparison studies for BLAS libraries. Finally, Section 7 summarizes the work with some conclusive remarks.

2. Background

2.1. Basic linear algebra subprograms

Basic Linear Algebra Subprograms (BLAS) provide routines as an Application Programming Interface (API) for performing vector and matrix operations. The BLAS routines are classified into three levels based on the degree of the polynomial in the complexities of operations:

Level 1 defines scalar, vector, and vector-vector operations. As an example, it includes a generalized vector addition (*axpy*, $a \times plus y$):

$$y \leftarrow ax + y \quad (1)$$

Level 2 defines matrix-vector operations. As an example, it includes a generalized matrix-vector multiplication (*gemv*):

$$y \leftarrow \alpha Ax + \beta y \quad (2)$$

Level 3 defines matrix-matrix operations. As an example, it includes a general matrix multiplication (*gemm*):

$$C \leftarrow \alpha AB + \beta C \quad (3)$$

Level 1 BLAS routines take linear time, $O(n)$, Level 2 routines quadratic time, and Level 3 routines cubic time [19]. Modern BLAS libraries provide routines from all three levels. The libraries include single-precision and double-precision versions for both real and complex numbers. While the earlier implementations only consider dense vectors and matrices, modern libraries also concentrate on sparse matrices.

The abstract definition of BLAS routines allows customization for high performance by exploiting architectural features in different computing systems. Due to the data-parallel nature of the

computations, especially SIMD execution units offer high-performance benefits. Both vector instructions in CPU systems and SIMD executions in GPU devices improve the performance of BLAS routines. While some BLAS libraries only include CPU-based implementations [20,21], others exploit parallel cores on GPU architectures as GPU devices become more available and popular [10,12,13].

In this work, we focus on Level 2 and Level 3 routines with higher complexities and evaluate the performance of GPU-based libraries by considering their GPU resource utilization.

2.2. GPU architectures

With a large number of processing units, GPU architectures offer high performance for applications from various domains. As illustrated in Figure 1, a modern GPU architecture contains hierarchical computational and memory units. Each core in a core cluster is responsible for single-instruction-multiple-thread (SIMT) execution. While the cores inside the same core cluster (Streaming multiprocessor-SM) have access to the scratchpad memory (shared memory or L1 cache), all the cores can communicate through the L2 cache structure via interconnect. DRAM-based global device memory maintains larger but relatively slower data access for all threads executing in the device [22]. Not only does a modern GPU device include general-purpose cores but also special function units (SFU) for fast transcendental function computations as well as tensor cores for efficient matrix multiplications. CUDA (Compute Unified Device Architecture) is a parallel programming model and application programming interface (API) that enables programmers to develop parallel software for general purpose processing on GPUs.

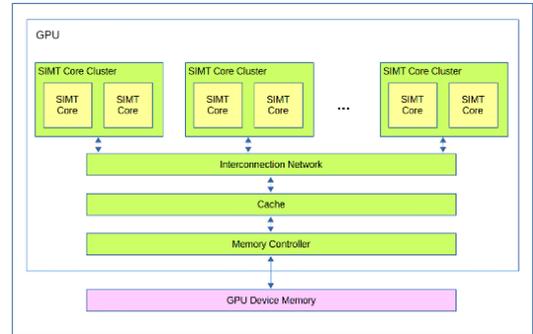


Figure 1. Modern GPU architecture.

3. Methodology

In this work, we evaluate two main BLAS libraries with GPU implementations. Specifically, we focus on Level 2 and Level 3 routines, which have higher computation requirements. To understand the GPU utilization of the target routines, we collect architectural performance metrics and compare the resource utilization of the most time-consuming parts of the computations. In our analysis, we investigate both SIMD unit and tensor core utilization and memory resource usage.

3.1. BLAS libraries

We consider two commonly used GPU-supporting BLAS libraries to examine in our performance evaluation study: cuBLAS [10] and MAGMA [13].

cuBLAS library, developed by NVIDIA, provides BLAS functions that are well-optimized for NVIDIA GPUs. Many machine learning frameworks (e.g., TensorFlow, Keras) rely on the cuDNN library [23], which utilizes the same kernel functions implemented as part of the cuBLAS library.

MAGMA (Matrix Algebra for GPU and Multicore Architectures) is an open-source project developed for hybrid multicore and multi-GPU systems. Based on LAPACK [24], which utilizes the shared-memory vector and parallel processors, MAGMA allows programmers to easily port their software from LAPACK to MAGMA to take advantage of the modern heterogeneous architectures.

Besides standard implementations, both libraries include batched operations by executing the multiple small kernels in parallel CUDA streams, multi-GPU executions for higher performance, and mixed and low precision executions offered by NVIDIA GPU functional units.

3.2. BLAS routines

While there are several BLAS routines in three levels (as given in Section 2.1), we focus on one main Level 2 function (GEMV) and five Level 3 functions (GEMM, SYMM, SYRK, TRMM, TRSM) in our evaluation. Since the computational complexity is high, especially in Level 3 matrix-matrix routines, we include those specific operations to examine their performance in our target massively parallel GPU device. The routines in our evaluation are as follows:

GEMV performs matrix-vector multiplication:

$$y = \alpha Ax + \beta y$$

where A is an m by n matrix, x and y are vectors, and α and β are scalars.

GEMM performs matrix-matrix multiplication:

$$C = \alpha AB + \beta C$$

where A , B , and C are an m by k , k by n , m by n matrices, and α and β are scalars.

SYMM performs symmetric matrix-matrix multiplication:

$$C = \alpha AB + \beta C$$

where A is an n by n symmetric matrix, B and C are m by n matrices, and α and β are scalars.

SYRK performs symmetric rank- k update to a matrix C :

$$C = \alpha A(A)^T + \beta C$$

where C is an n by n symmetric matrix, A is an n by k matrix, and α and β are scalars.

TRMM performs triangular matrix-matrix multiplication, where one input matrix is triangular, and one input matrix is general, and **TRSM** performs solving triangular matrix with multiple right-hand sides.

We evaluate both single-precision and double-precision functions provided by cuBLAS and MAGMA libraries. Additionally, for GEMM operation, we execute the half-precision routines, which utilize tensor cores in the target GPU device.

3.3. Performance metrics

To understand the performance of target operations running on GPU architectures and perform a comparison study, we consider not only kernel execution times but also architectural metrics provided by profiling tools. Since we are interested in performance analysis on GPU devices at the chip level to see if any specific architecture features are responsible for the performance difference, we collect information for the kernel executions. We do not include the data transfer time from CPU memory to GPU global memory (HtoD) or GPU global memory to

CPU memory (DtoH). The performance metrics collected from kernel executions and profiling tools are as follows:

FLOPS: Since BLAS routines contain intensive floating-point calculations, we report floating-point operations per second (FLOPS), which is a measure of computer performance.

Compute resource utilization: We analyze how the target routines utilize the compute resources available in GPU devices. We consider compute resources of the streaming multiprocessors (SM) by including both high-level SM utilization values and detailed pipeline utilization values per each pipeline, such as LSU (Load-Store unit) and ALU (Arithmetic logic unit).

Memory resource utilization: We analyze how the target routines utilize the memory resources available in GPU devices. While GPU architectures include a hierarchical memory structure with different levels, we consider the throughput values for global memory. Moreover, our pipeline utilization analysis demonstrates the usage of load-store units, which is higher for memory-bound operations.

Warp occupancy: Warp occupancy is a measure of thread parallelism in a GPU program, which is defined as the number of warps running concurrently on a multiprocessor divided by the maximum number of warps that can run concurrently. Since the available registers and shared memory are shared among all active warps on a streaming multiprocessor, the number of active warps can be limited by the register and shared memory usage of the threads. The higher register or shared memory usage in each thread limits the number of active warps running simultaneously on a streaming multiprocessor. While higher occupancy does not always reveal higher performance, occupancy values provide information about the device limitations and guide the programmer for register/memory usage.

Table 1. Salient characteristics of GPU device used in our experiments.

| Property | Value |
|--------------------------|---------------|
| CUDA Compute Capability | 7.5 |
| Global memory size | 3914 MB |
| Multiprocessors | 14 MP |
| CUDA cores per MP | 64 |
| GPU Max Clock rate | 1455 MHz |
| Memory Clock rate | 4001 MHz |
| Memory Bus Width | 128-bit |
| L2 Cache Size | 1048576 bytes |
| Max Warps per MP | 32 |
| Max Thread Blocks per MP | 16 |
| Max Threads per MP | 1024 |
| Registers per MP | 65536 |
| Shared memory per MP | 65536 |

4. Performance Evaluation

We evaluate the performance of different routines in cuBLAS and MAGMA libraries on our mobile workstation with an NVIDIA T1000 GPU device, a Turing architecture [25] professional

Table 2. Properties of kernel functions in single-precision routines.

| BLAS routine | Function name | Kernel name | Grid size | Block size | Threads | Cycles |
|--------------|------------------------|---------------------------------------|-----------|------------|-----------|-------------|
| GEMV | <i>magmablas_sgemv</i> | <i>sgemvn_template_kernel_fermi</i> | 98 | 512 | 50,176 | 482,099 |
| | <i>cublasSgemv</i> | <i>gemv2N_kernel</i> | 400 | 128 | 51,200 | 479,169 |
| GEMM | <i>magmablas_sgemm</i> | <i>sgemm_kernel_fermi_nn</i> | 1,089 | 256 | 278,784 | 56,212,094 |
| | <i>cublasSgemm</i> | <i>volta_sgemm_128x128_nn</i> | 625 | 256 | 160,000 | 37,636,048 |
| SYMM | <i>magmablas_ssym</i> | <i>hemm_template_ll_kernel</i> | 9,604 | 256 | 2,458,624 | 182,148,002 |
| | <i>cublasSsym</i> | <i>magma_lds128_strmm_kernel (2)</i> | 2,401 | 128 | 307,328 | 30,964,954 |
| SYRK | <i>magmablas_ssyрк</i> | <i>volta_sgemm_64x32_sliced1x4_nt</i> | 656 | 256 | 167,936 | 5,889,255 |
| | <i>cublasSsyрк</i> | <i>volta_sgemm_128x128_lower_nt</i> | 625 | 256 | 160,000 | 19,417,160 |
| TRMM | <i>magmablas_strmm</i> | <i>trmm_template_lNx_kernel (98)</i> | 98 | 1024 | 100,352 | 34,142 |
| | <i>cublasStrmm</i> | <i>trmm_left_kernel_core (24)</i> | 98 | 512 | 50,176 | 262,627 |
| TRSM | <i>magmablas_strsm</i> | <i>volta_sgemm_128x32_nn (31)</i> | 98 | 256 | 25,088 | 106,655 |
| | <i>cublasStrsm</i> | <i>volta_sgemm_32x128_nn (5)</i> | 850 | 256 | 217,600 | 13,963,232 |

mobile graphics card. Our target GPU device, of which the main features are given in Table 1, can optimize both single-precision and double-precision calculations and perform tensor operations. Our platform runs on Ubuntu 18.04 operating system, with MAGMA version 2.6.2 and cuBLAS version 11.3. We utilize single-precision and double-precision versions available in the libraries for BLAS routines. Additionally, we execute half-precision GEMM routines, which utilize FP16 tensor functional units in our GPU device.

Firstly, we execute our target BLAS routines and collect GFLOPS (billion floating-point operations per second) values for the entire routine execution. Specifically, our experiments consist of ten executions, where we report the average.

Then, we run the same configurations on NVIDIA Nsight Compute [26], a kernel profiler for CUDA applications, and collect detailed performance metrics for each kernel function. Nsight Compute provides detailed performance metrics and visual representations for resource usage of CUDA kernels. Based on the performance metrics explained in the previous section, we examine compute and memory resource utilization by considering hardware limitations. We analyze the performance and resource utilization of the implementations available in both libraries by discussing the potential bottlenecks.

We note that we perform a precision comparison for the computations by comparing CPU results and observe a maximum error of less than $e-08$.

4.1. Kernel functions in single-precision and double-precision routines

We execute both single-precision and double-precision BLAS routines from cuBLAS and MAGMA libraries. The corresponding functions in the libraries follow a standard naming, such as cuBLAS has *cublasSgemm* and *cublasDgemm*, MAGMA has *magmablas_sgemm* and *magmablas_dgemm* for single-precision and double-precision GEMM computations, respectively.

Using the Nsight Compute tool, we identify the corresponding kernel functions and collect their runtime properties, including

thread dimensions and the number of cycles. As a representative input size, we select 3136 as the matrix/vector dimension and collect performance metrics for this input. Table 2 and Table 3 present the launch configuration parameters and the cycles for each kernel in the BLAS routines. While some routines include only one kernel function (e.g., *sgemm_kernel_fermi_nn* for single-precision GEMM routine in MAGMA), the others consist of several kernel functions in different sizes in terms of thread sizes and execution time. In the case of multiple kernel functions, either several light-weight kernels are executed and perform partial computation or one heavy kernel (e.g., *trmm_left_kernel_core* in *cublasStrmm* routine) is launched multiple times and performs the main computation.

In Table 2 and Table 3, we put the largest/dominant kernel functions from each BLAS routine (and the number of kernel launches in parenthesis if launched more than once). For instance, there are 18 *volta_dgemm_64x64_nn* instances in *cublasDtrmm* routine (as given in Table 3), each launched with 50,176 threads (784 blocks and 64 threads per block), and each takes ~ 120 million cycles. Since multiple instances may take different cycles, we report the largest instance in tables to include the most heavy-weight kernel instance. For further investigation, one can examine the execution of this specific instance.

4.2. GFlops

Before examining the architectural properties and GPU resource usage of the target BLAS computations, we execute all the routines with ten different input sizes and collect Gflop values for the complete execution. We generate the vectors and matrices randomly to represent dense structures. Figure 2 presents Gflop values for single-precision and double-precision routines.

We observe substantial differences between cuBLAS and MAGMA single-precision computations. cuBLAS exhibits much larger Gflop values, especially for SSYMM and SSYRK operations. On the other hand, the Gflop values are much closer for double-precision computations, where the pressure on computational resources is higher due to higher precision.

Table 3. Properties of kernel functions in double-precision routines.

| BLAS routine | Function name | Kernel name | Grid size | Block size | Threads | Cycles |
|--------------|-----------------|------------------------------|-----------|------------|-----------|---------------|
| GEMV | magmablas_dgemv | dgemvn_template_kernel_fermi | 98 | 512 | 50,176 | 947,981 |
| | cublasDgemv | gemvNSP_kernel | 196 | 512 | 100,352 | 957,724 |
| GEMM | magmablas_dgemm | dgemm_kernel_fermi_nn | 2,401 | 256 | 614,656 | 1,117,379,580 |
| | cublasDgemm | volta_dgemm_128x64_nn | 2,450 | 128 | 313,600 | 1,130,253,310 |
| SYMM | magmablas_dsym | hemm_template_ll_kernel | 9,604 | 256 | 2,458,624 | 1,112,063,431 |
| | cublasDsym | magma_lds128_dtrmm_kernel | 4,802 | 128 | 614,656 | 551,819,169 |
| SYRK | magmablas_dsyrk | volta_dgemm_128x64_nt (7) | 600 | 128 | 76,800 | 185,849,226 |
| | cublasDsyrk | volta_dgemm_128x64_lower_nt | 1,225 | 128 | 156,800 | 592,198,360 |
| TRMM | magmablas_dtrmm | volta_dgemm_64x64_nn (91) | 196 | 64 | 12,544 | 7,457,654 |
| | cublasDtrmm | volta_dgemm_64x64_nn (18) | 784 | 64 | 50,176 | 120,024,331 |
| TRSM | magmablas_dtrsm | volta_dgemm_12_x64_nn (20) | 1,127 | 128 | 144,256 | 43,314,307 |
| | cublasDtrsm | volta_dgemm_64x64_nn (6) | 833 | 64 | 53,312 | 252,359,168 |

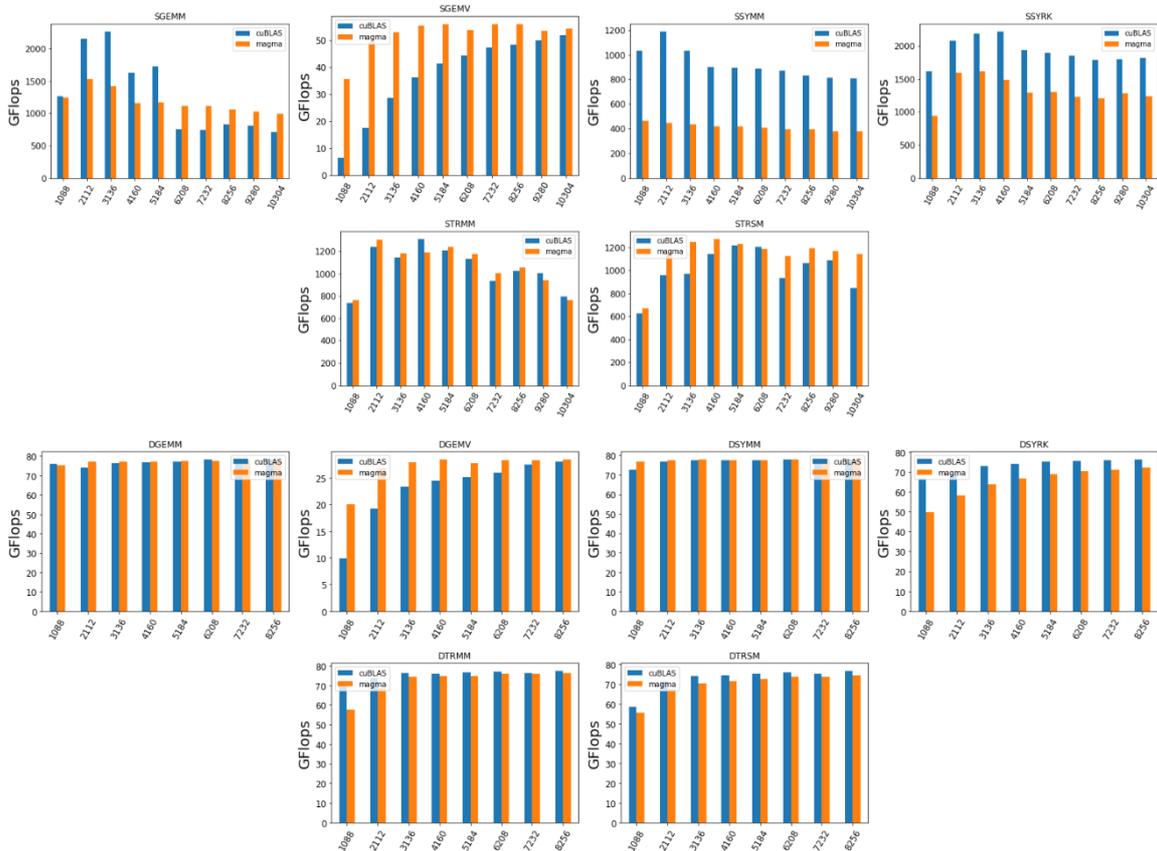


Figure 2. GFlop values for single-precision and double-precision routines.

4.3. Resource utilization

We collect SM and memory utilization values to explain the performance differences (GFlop values) observed in the previous section. As stated before, we analyze one instance of the kernel functions with the largest total number of cycles (given in Table

2 and Table 3). Essentially, all instances of the functions demonstrate similar utilization values as the indication of the performance bottleneck.

Figure 3 presents both SM and memory utilization values (out of 100%) for the kernel functions in our consideration. While

Table 4. Occupancy metrics for double-precision GEMM kernels.

| | cuBLAS | MAGMA |
|---|--------------------------|--------------------------|
| Registers per Thread | 234 | 61 |
| Shared memory per Block (bytes) | 25088 | 17024 |
| Threads per Block | 128 | 256 |
| Active Threads per SM Limit (Register) | 256 ($\leq 65536/234$) | 1024 ($\leq 65536/61$) |
| Active Thread Blocks per SM Limit (S. Memory) | 2 ($\leq 65536/25088$) | 3 ($\leq 65536/17024$) |
| Active Thread Blocks-Threads per SM | 2-256 (2x128) | 3-768 (3x256) |
| Theoretical Occupancy for Each SM | 25% (256/1024) | 75% (768/1024) |
| Achieved Occupancy | 24.93% | 74.65% |

computationally-intensive Level-3 computations demonstrate very high SM utilization values (larger than 90%), especially for double-precision operations, the memory is the bottleneck for intensive Level-2 *GEMV* routine.

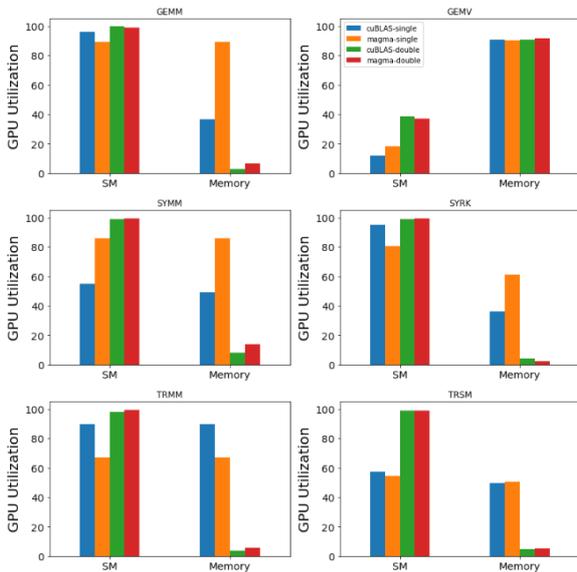


Figure 3. Utilization values for kernel functions in single-precision and double-precision routines.

Furthermore, we closely look at the compute pipeline utilization to understand how the kernel functions make use of the functional units in GPU cores. Figure 4 presents the utilization for the compute resources of the streaming multiprocessors. In our diagrams, we include only the highest compute pipelines. Mainly, single-precision routines utilize LSU and FMA units, and double-precision routines utilize LSU and FP64 units. LSU stands for Load Store Unit, which issues load, store, atomic, and reduction instructions to the first-level cache for memory access operations. FMA stands for Fused Multiply Add/Accumulate Unit, which performs FP32 arithmetic (Single-precision floating-point format), including FADD, FMUL, and FMAD. FP64 represents double-precision floating-point unit, which is responsible for FP64 arithmetic (Double-precision floating-point format), including DADD, DMUL, and DMAD. Additionally, we observe ALU unit (Arithmetic Logic Unit) utilization, which performs bit manipulation and logic instructions as well as integer (i.e., IMAD, IMUL) operations. Since ALU unit utilization gets lower than 20%

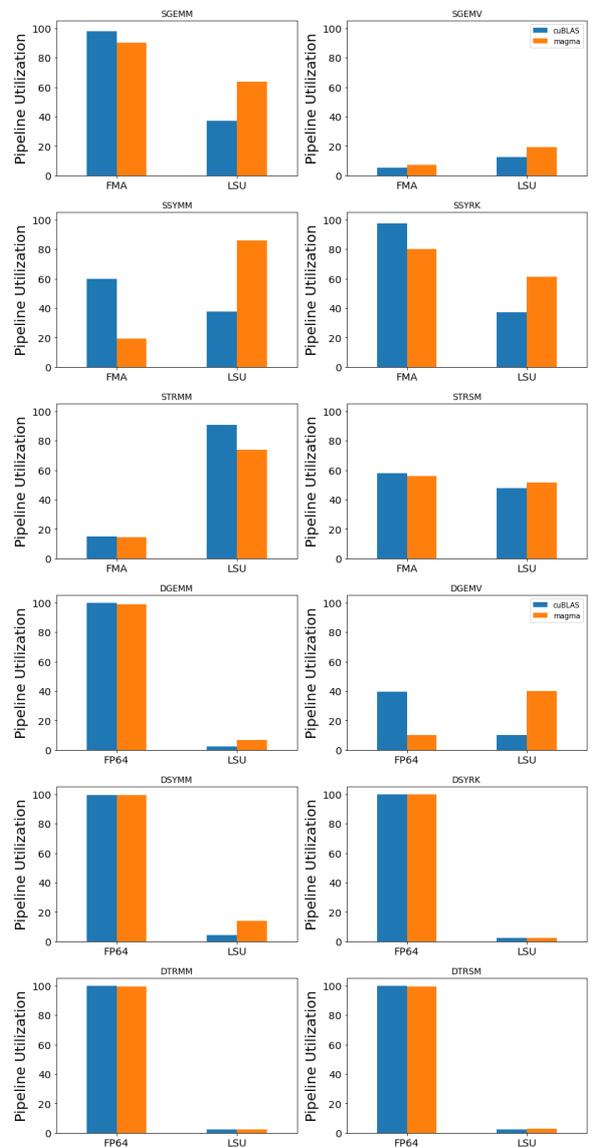


Figure 4. Pipeline utilization values for single-precision and double-precision routines.

for most cases and does not give insights about our analysis, we do not report its values.

While we report the utilization values for both single-precision and double-precision computations, we focus on the consistent double-precision values. In parallel to SM and memory utilization values, Level-3 double-precision routines exhibit large FP64 utilization by spending most of their time in high-precision floating-point computations. Since double-precision Level-3 BLAS routines require high computation resources, we observe that they consume all available GPU multiprocessors/cores in our device with almost 100% SM utilization values.

4.4. Warp occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. The higher occupancy does not always mean higher performance, but reveals the ability of GPU device to have active warps. Even if the number of threads is large and GPU device offers a large number of cores, the occupancy can be low due to the requirements of individual threads, such as number of registers or shared memory.

We collect occupancy results from NVIDIA Nsight Compute profiles, which provides a set of metrics that identifies occupancy. *Theoretical Occupancy* represents the upper limit for occupancy due to the kernel launch configuration and the GPU device capabilities. *Achieved Occupancy* measures the occupancy during execution of the kernel. Additionally, we collect details about the occupancy-limiter factors. Moreover, we focus on double-precision GEMM routines since they exhibit more consistent and reasonable executions.

Table 4 presents occupancy limiters and occupancy values for GEMM kernels in both libraries. We see 25% and 75% occupancy values for cuBLAS and MAGMA kernels, respectively. By considering the upper limits of our GPU device multiprocessor (32 warps, 16 thread blocks, 1024 threads, 65536 registers, and 65536 bytes shared memory per SM as given in Table 1), we can have 256 active threads (8 warps) and 768 active threads (24 warps) per SM while it is possible to execute 1024 threads (32 warps). While the register and shared memory usage of cuBLAS threads limit the number of active warps, the shared memory usage is the limitation for the MAGMA kernel execution.

Even if the kernel functions have diverse occupancy values and cuBLAS seems to be inefficient in terms of occupancy, we observe similar performance in terms of cycles (Table 3), GFlop values (Figure 2), and SM utilization (Figure 3). cuBLAS threads, which utilize low-latency memory structures like registers and shared memory, execute faster even if the execution does not involve many active threads. On the other hand, MAGMA kernel with more threads utilizes SM resources efficiently and benefits from the parallel cores in the GPU multiprocessor. Additionally, we observe no significant difference between theoretical and achieved occupancy values in the kernel, which emphasizes no serious imbalance issue in the kernel executions [27].

4.5. Tensor core usage

NVIDIA has introduced Tensor Cores in 2017 with Volta V100 GPUs [28] to accelerate matrix multiplication operations with dedicated hardware. NVIDIA Tensor Cores present the native instructions for half-precision matrix multiply operations [29]. They offer shorter execution times for linear algebra methods, which include matrix operations, by utilizing mixed-precision arithmetic [30].

Both cuBLAS and MAGMA libraries offer Half-precision General Matrix Multiply (HGEMM) routines that enable tensor core

utilization: *cublasHgemm* and *magma_hgemm*. To observe the performance of tensor cores, we utilize half-precision GEMM routines that perform the operations in the tensor core's 16-bit execution units.

We execute and profile half-precision routines from the libraries and observe that both routines rely on CUTLASS [31], which is a collection of CUDA C++ template abstractions for implementing GEMM operations within CUDA. From our profiling results obtained from NVIDIA Nsight Compute, we observe that both cuBLAS and MAGMA executions utilize the same CUTLASS function, therefore, perform similarly by utilizing Tensor (FP) units. Since cuBLAS execution utilizes the same CUTLASS kernel function for the target execution, we only report the MAGMA values to avoid duplication. Figure 5 presents normalized GFlop values for half-precision GEMM routine (HGEMM) compared to double-precision GEMM (DGEMM) with different matrix dimensions. Both precision reduction in function arguments and tensor core usage in computation reduce the computation time for various input sizes. Additionally, Figure 5 presents SM and memory utilization values collected from NVIDIA Compute for 3136 input size. As discussed in Section 4.3, double-precision GEMM computations require high computational resources. SM utilization is the bottleneck, with almost 100% SM usage. Hence, they are compute-bound. On the other hand, half-precision GEMM computations neither fully utilize SM cores nor memory bandwidth for the same number of elements. Since Tensor cores maintain high performance due to their half-precision specialized units, they could offer scalable high performance [32,33].

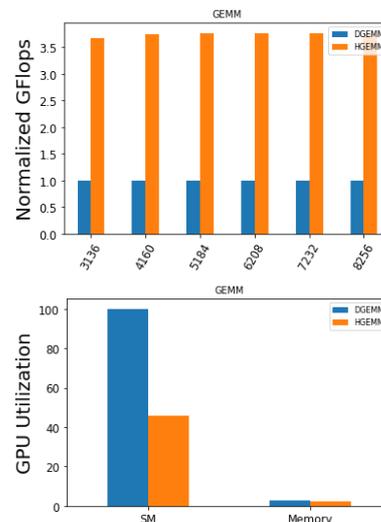


Figure 5. Normalized GFlop and utilization values for double-precision vs half-precision.

5. Discussion

While double-precision operations and Level-3 routines with larger computations utilize computation resources efficiently, the memory operations become the bottleneck for single-precision operations and Level-2 routines with less computations. If we compare cuBLAS and MAGMA routines, they do not exhibit much difference at computationally-heavy operations; however, cuBLAS offers higher performance in terms of GFlops and higher SM utilization for most of the executions. One can prefer cuBLAS for small-scale operations, if the computational resources in the target GPU device are relatively limited. On the other hand, for large-scale DNN operations, which include heavy GEMM computations, the performance mainly depends on the parallelism support of the target device other

Table 5. Performance comparison studies for BLAS libraries.

| Work | Libraries | Operations | Metrics |
|----------------------|---------------------------------------|--|--|
| Dongarra et al. [16] | MKL, OMP (CPU) CuBLAS, MAGMA (GPU) | Batched GEMM | Gflop/s |
| Li et al. [17] | MKL (CPU), CuBLAS (GPU) | DOT, GEMV, GEMM, TRSV, and TRSM | Execution time |
| Ganeshan et al. [18] | MAGMA and CuBLAS (GPU) | Vector Fitting (QR decomposition) | Execution time |
| Our Work | MAGMA and CuBLAS (GPU) | All Level-3 and one Level-2 BLAS routines | Execution cycles, GFlop, SM, memory, pipeline utilization, occupancy |

than the underlying BLAS library implementation. Especially, the tensor core execution support, which is specialized for matrix multiplication operations, gets a key issue to accelerate GEMM operations.

6. Related Works

BLAS libraries have been utilized by neural network implementations and general-purpose applications. Additionally, there are studies comparing CPU or GPU performance of BLAS libraries.

Dongarra et al. [16] propose a block-interleaved approach for batched DGEMM operations to improve performance by considering the data layout of the matrices in the system memory. The authors perform a comparison study by considering batched GEMM operations, which include multiple BLAS operations in parallel on many small matrices, with MKL, OpenMP, CuBLAS, and MAGMA libraries, and analyze the impact of the data layout on the performance.

Li et al. [17] present a comparison study between a subset of BLAS operations from MKL library running on a multi-core CPU system and cuBLAS library utilizing a many-core GPU architecture. The authors summarize the implementation and parallelization details and conduct a performance evaluation by considering execution time for a variety of matrix sizes.

Ganeshan et al. [18] focus on GPU execution of vector fitting (GVF) algorithm. After presenting mathematical representation of the algorithm, the authors execute the codes that are based on QR decomposition operations. They utilize *magma_dgeqrf_batched* and *cublas_dgeqrf_batched* routines for the corresponding functionality from MAGMA and cuBLAS libraries, respectively. The results, based on execution times, demonstrate higher performance for MAGMA-based execution for the target GVF algorithm.

Although the previous studies perform comparison analyses by considering CPU and GPU performance of BLAS libraries, we focus on GPU executions and perform a detailed performance analysis for computationally-heavy BLAS routines by considering architectural metrics and resource utilization. Table 5 summarizes the related performance comparison work and our work based on the differences including target BLAS libraries, BLAS operations, and performance metrics used in the comparison study.

7. Conclusion

In this study, we examine the architectural characteristics of BLAS routines implemented in cuBLAS and MAGMA libraries. We execute target routines from both libraries in our target GPU

device and evaluate performance metrics by considering architectural resource utilization and limitations. We believe that our quantitative analysis will help researchers and programmers who utilize library functions to understand target executions and make decisions.

Our work can be expanded by comparing the BLAS libraries in different GPU device generations to better understand the effect of the architectural differences. Additionally, the implementations of the domain-specific applications based on different BLAS libraries can be evaluated by comparing the resource utilization of the target application.

Ethics committee approval and conflict of interest statement

This article does not require ethics committee approval.

This article has no conflicts of interest with any individual or institution.

References

- [1] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, J. D. Owens, 2016. Gunrock: A high-performance graph processing library on the gpu, ACM SIGPLAN Notices, 51 (8), 1–12. DOI: 10.1145/3016078.2851145
- [2] S. Le Grand, A. W. Götz, R. C. Walker, 2013. Spfp: Speed without compromise—a mixed precision model for gpu accelerated molecular dynamics simulations, Computer Physics Communications 184 (2), 374–380. DOI: 10.1016/j.cpc.2012.09.022
- [3] A. Zeni, G. Guidi, M. Ellis, N. Ding, M. D. Santambrogio, S. A. Hofmeyr, A. Buluc, L. Oliker, K. A. Yelick, 2020. LOGAN: high-performance gpu-based x-drop long-read alignment, IEEE International Parallel and Distributed Processing Symposium (IPDPS).
- [4] F. F. d. Santos, P. F. Pimenta, C. Lunardi, L. Draghetto, L. Carro, D. Kaeli, P. Rech, 2019. Analyzing and increasing the reliability of convolutional neural networks on gpus, IEEE Transactions on Reliability 68 (2), 663–677. DOI: 10.1109/TR.2018.2878387
- [5] S. Alcaide, L. Kosmidis, H. Tabani, C. Hernandez, J. Abella, F. J. Cazorla, 2018. Safety-related challenges and opportunities for gpus in the automotive domain, IEEE Micro 38 (6), 46–55. DOI: 10.1109/MM.2018.2873870
- [6] M. Benito, M. M. Trompouki, L. Kosmidis, J. D. Garcia, S. Carretero, K. Wenger, 2021. Comparison of gpu computing methodologies for safety-critical systems: An avionics case study, Design, Automation Test in Europe Conference Exhibition (DATE).
- [7] S. Kestur, J. D. Davis, O. Williams, 2010. Blas comparison on fpga, cpu and gpu, IEEE Computer Society Annual Symposium on VLSI.
- [8] A. A. Awan, H. Subramoni, D. K. Panda, 2017. An in-depth performance characterization of cpu- and gpu-based dnn training on modern architectures, Proceedings of the Machine Learning on HPC Environments (MLHPC).
- [9] A. Abdelfattah, D. Keyes, H. Ltaief, 2016. Kblas: An optimized library for dense matrix-vector multiplication on gpu accelerators, ACM Trans.Math. Softw. 42 (3), 1–31. DOI: 10.1145/2818311
- [10] cublas: Basic linear algebra on nvidia gpus. <https://developer.nvidia.com/cublas> (Access Date: January 2023).
- [11] C. Brown, A. Abdelfattah, S. Tomov, J. Dongarra, 2020. Design, optimization, and benchmarking of dense linear algebra algorithms on

- amd gpus, IEEE High Performance Extreme Computing Conference (HPEC).
- [12] roclblas user guide. <https://roclblas.readthedocs.io/> (Access Date: January 2023).
- [13] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, 2014. Accelerating numerical dense linear algebra calculations with gpus, *Numerical Computations with GPUs*, Springer, Cham.
- [14] L. Wang, W. Wu, Z. Xu, J. Xiao, Y. Yang, 2016. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing, *Proceedings of the International Conference on Supercomputing*, (ICS).
- [15] C. Nugteren, 2018. Cblast: A tuned opencl blas library, *International Workshop on OpenCL (IWOCCL)*.
- [16] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, M. Zounon, 2017. The design and performance of batched blas on modern high-performance computing systems, *Procedia Computer Science* 108, 495–504. DOI: 10.1016/j.procs.2017.05.138
- [17] F. Li, Y. Ye, Z. Tian, X. Zhang, 2019. CPU versus GPU: which can perform matrix computation faster - performance comparison for basic linear algebra subprograms, *Neural Comput. Appl.* 31 (8), 4353–4365. DOI: 10.1007/s00521-018-3354-z
- [18] S. Ganeshan, N. K. Elumalai, R. Achar, 2020. A comparative study of magma and cublas libraries for gpu based vector fitting, *IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*.
- [19] J. J. Dongarra, J. Du Croz, S. Hammarling, I. S. Duff, 1990. A set of level 3 basic linear algebra subprograms, *ACM Transactions on Mathematical Software*, 16 (1), 1–17 16 (1). DOI: 10.1145/77626.79170
- [20] Z. Xianyi, M. Kroeker, Openblas: An optimized blas library. <https://www.openblas.net/> (Access Date: January 2023).
- [21] R. Clint Whaley, A. Petitot, J. J. Dongarra, 2001. Automated empirical optimizations of software and the atlas project, *Parallel Computing* 27 (1), 3–35. DOI: 10.1016/S0167-8191(00)00087-9
- [22] T. M. Aamodt, W. W. L. Fung, T. G. Rogers, M. Martonosi, 2018. *General-Purpose Graphics Processor Architecture*, Morgan and Claypool Publishers.
- [23] Nvidia cudnn. <https://developer.nvidia.com/cudnn> (Access Date: January 2023).
- [24] Lapack-linear algebra package. <http://www.netlib.org/lapack/> (Access Date: January 2023).
- [25] Nvidia-turing architecture white paper. <https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf> (Access Date: May 2022).
- [26] Nvidia nsight compute. <https://developer.nvidia.com/nsight-compute> (Access Date: January 2023).
- [27] M. Awatramani, X. Zhu, J. Zambreno, D. Rover, 2015. Phase aware warp scheduling: Mitigating effects of phase behavior in gpgpu applications, *International Conference on Parallel Architecture and Compilation (PACT)*.
- [28] Nvidia tesla v100 gpu architecture. <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf> (Access Date: May 2022).
- [29] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, J. S. Vetter, 2018. NVIDIA tensor core programmability, performance and precision, *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*.
- [30] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojear, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, U. M. Yang, 2021. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic, *The International Journal of High Performance Computing Applications* 35 (4), 344–369. DOI: 10.1177/10943420211003313
- [31] Cutlass. <https://github.com/NVIDIA/cutlass> (Access Date: January 2023).
- [32] A. Abdelfattah, S. Tomov, J. Dongarra, 2019. Towards half-precision computation for complex matrices: A case study for mixed precision solvers on gpus, *IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (Scala)*.
- [33] D. Yan, W. Wang, X. Chu, 2020. Demystifying tensor cores to optimize half-precision matrix multiply, *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [34] X. Li, G. Zhang, H. H. Huang, Z. Wang and W. Zheng, 2016. Performance Analysis of GPU-Based Convolutional Neural Networks, *45th International Conference on Parallel Processing (ICPP)*.
- [35] Jon Perez-Cerrolaza, Jaume Abella, Leonidas Kosmidis, Alejandro J. Calderon, Francisco Cazorla, and Jose Luis Flores, 2023. GPU Devices for Safety-Critical Systems: A Survey. *ACM Comput. Surv.* 55, 7, Article 147. DOI: 10.1145/3549526
- [36] Pandey, M., Fernandez, M., Gentile, F. et al. 2022. The transformational role of GPU computing and deep learning in drug discovery. *Nat Mach Intell* 4, 211–221. DOI: 10.1038/s42256-022-00463-x
- [37] Sergio Barrachina, Manuel F. Dolz, Pablo San Juan, Enrique S. Quintana-Orti, 2022. Efficient and portable GEMM-based convolution operators for deep neural network training on multicore processors, *Journal of Parallel and Distributed Computing*, 167, 240-254. DOI: 10.1016/j.jpdc.2022.05.009
- [38] Susmita Dey Manasi, Suvadeep Banerjee, Abhijit Davare, Anton A. Sorokin, Steven M. Burns, Desmond A. Kirkpatrick, and Sachin S. Sapatnekar, 2023. Reusing GEMM Hardware for Efficient Execution of Depthwise Separable Convolution on ASIC-Based DNN Accelerators. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASPDAC)*.